Stegasaurus: A Simple Image Steganography Program

Kendall Stewart
CS 557: Functional Programming
Portland State University

March 17, 2014

1 Introduction

Steganography is the art of disguising covert messages as genuine articles in some sort of cover medium. As a practice, it can be traced back to ancient times [1]. Many different cover media and transmission methods can be imagined – perhaps a slight variation of letterforms in a handwritten missive, or a particular choice of steps at a formal dance. Even a subtle cough at the right moment could be considered a form of steganography. What all these things have in common is that both the sender and the receiver must be aware of the protocol, and that the message must appear "invisible" to anyone not aware of the protocol.

In recent years, the most common media for steganographic message hiding have been forms of digital information, transmitted over the internet. Many novel media have been discovered, including text, images, video, and music, as well as some more exotic ones such as social media sites, filesystems or network protocol packets [1].

Because steganography is most concerned with avoiding detection, it is often abused for illicit purposes [1]. However, steganography has some legitimate uses, most notably for circumventing online censorship. Censorship circumvention is becoming increasingly important; Tom Shrimpton, a Professor at Portland State University, was recently awarded a surprise grant by Google for his research in that field [2]. While his research was not in specifically in steganography, a paper by Shrimpton et. al. acknowledges that steganography is often used for the same purpose [3].

This report details the construction of a tool (written in the Haskell language) for hiding a textual message in a digital image file. Along the way, we provide some light analysis of its performance in the context of image processing and steganography in general.

2 The Method

The overall approach used in this program is relatively straightforward: each character in a message composed of 7-bit ASCII characters is encoded into a single pixel in a 24-bit RGB color image, via bit manipulation. Pixels to be "commandeered" are selected based on the number of characters in the message.

2.1 Encoding a single character

Each character in the message is encoded in a 24-bit RGB pixel, as follows: The 7-bit ASCII value is divided into three parts. Encoding is done simply by overwriting the low-order bits of each pixel component with the appropriate section of the character. The two highest order bits are given the red component, the next two bits to the green component, and the low three order bits to the blue component.

An illustration of this process is given in figure 1.

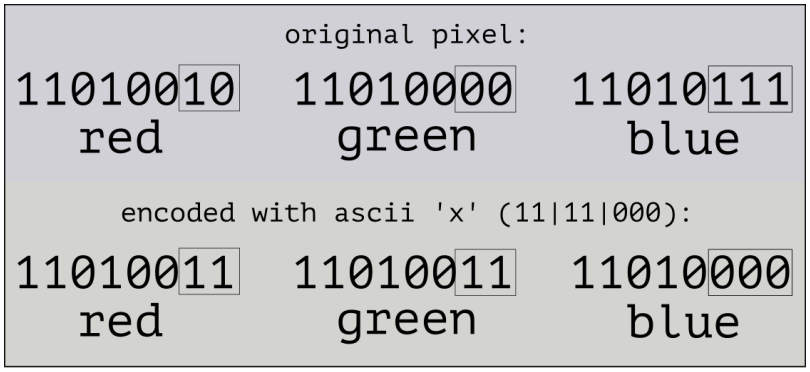

Figure 1: An illustration of the encoding method for a single pixel.

To recover the character from an encoded pixel, the high order bits of each component are simply thrown away, and the character bits are rejoined. Haskell code for carrying out the encoding and decoding process is given below:

```
data Pixel = Pixel {r::Int, g::Int, b::Int} deriving (Show, Eq)

encodePixel :: Pixel -> Int -> Pixel
encodePixel (Pixel baseR baseG baseB) key
  = Pixel (z 2 baseR .|. rBits) (z 2 baseG .|. gBits) (z 3 baseB .|. bBits)
  where z n b = b `shiftR` n `shiftL` n
        rBits = (key `shiftR` 5) .&. mask 2
        gBits = (key `shiftR` 3) .&. mask 2
        bBits =  key             .&. mask 3

decodePixel :: Pixel -> Int
decodePixel (Pixel encR encG encB)
  = rBits .|. gBits .|. bBits
  where rBits = (encR .&. mask 2) `shiftL` 5
        gBits = (encG .&. mask 2) `shiftL` 3
        bBits =  encB .&. mask 3

mask  :: Int -> Int
mask n = 2^n - 1
```

This code is relatively straightforward – during encoding, the low-order component bits are zeroed out by shifting right, then left. The appropriate message bits are selected with right-shifting and masking, and then dropped into the component with an `OR` operation. During decoding, the message bits are recovered and then joined with an `OR` operation.

2.2 Encoding the whole message

Next we turn to the task of encoding an entire message. One simple method would be to simply start at the beginning of the image, encoding each pixel in turn until the message is exhausted, then encode some sort of "end of file" sentinel value. That way, when recovering the message, we would know when to stop decoding pixels. This would be easy to implement, but it has the disadvantage of bunching all of the message pixels at the top of the image, which may be easier to detect.

The method we use instead is to distribute message pixels throughout the image, by dividing the available pixels by the length of the message, and then only "sprinkling" message characters at that regular interval. This requires a little bit more machinery to implement, as now we need a way of performing this selective alteration. On top of that, in order to recover a particular message, we need to be able to recover the spacing interval.

In order to accomplish this, we encode the length of the message in the first three pixels of the image. Given that 7 bits per pixel are used for encoding, this affords us a maximum message length of $2^{(7 \cdot 3)} - 1 \approx 2.1$ million characters.

We define two functions for handling the length decoding/encoding, as well as a constant for the maximum length (it will be used in other places as well):

```
maxLength     = 2^21 - 1

encodeLength :: Int -> [Pixel] -> [Pixel]
decodeLength :: [Pixel] -> Int
```

The implementation of these functions can be found in the appendix.

Now we turn to the question of how to evenly distribute the message throughout the image. First we must choose a representation of images. We use a simple data structure that contains the height, the width, and the image data (as a list of lists of pixels):

```
data Image = Image {height :: Int, width :: Int, pixels :: [[Pixel]]}
  deriving (Show, Eq)
```

It is also helpful to define a function for giving us the number of pixels in an image available for encoding (which is all of them, except the ones used for the length):

```
headerLength = 3
availPix    :: Image -> Int
availPix img = height img * width img - headerLength
```

We then take the following approach. At each pixel in the image, examine its index (starting from zero). If the index is aligned with the spacing, encode a character from the message. Otherwise, do nothing. This approach suggests the need to maintain some state information: the current index, the remaining message characters, and so forth.

To that end, we introduce a simple state manipulation monad [4], as well as a `Sprinkler` datatype to store the relevant state information:

```
newtype ST s a = ST {run :: s -> (a, s)}

instance Monad (ST s) where
  return x = ST (\s -> (x, s))
```

```
  c >>= f  = ST (\s -> let (x, s') = run c s in run (f x) s')

upd  :: (a -> a) -> ST a a
upd f = ST (\s -> (s, f s))

data Sprinkler = Sprinkler { index    :: Int,
                             spacing  :: Int,
                             message  :: String,
                             currlen  :: Int,
                             msglen   :: Int }
```

We then define a function to construct a `Sprinkler` value from an image and a message:

```
genSprinkler :: Image -> String -> Sprinkler
```

As well as a function to update the state of the `Sprinkler` as we move from one pixel to the next:

```
advance      :: Sprinkler -> Sprinkler
```

The implementation of these two helper functions can be found in the appendix.

Now we have enough to create a `sprinkle` function which takes a message, an image, and then returns an image (in case of success) or nothing (in case of failure). Failure occurs when the message is too long for the image; this is either because the message exceeds the maximum length, or because the message has more characters than there are available pixels in the image.

A helper function, `sprinkle'`, performs the actual task of distributing the message throughout the file, using the monadic state. At each pixel, it updates the state, and then checks to see if a message character is meant to be "dropped" at this pixel. If so, it encodes the pixel with the current character.

```
sprinkle :: String -> Image -> Maybe Image
sprinkle message image
  | len > availPix image || len > maxLength = Nothing
  | otherwise                               = Just image{pixels = group w modified}
    where modified     = header ++ (fst (run (sprinkle' rest) sprinkler))
          sprinkler    = genSprinkler image message
          imgVec       = concat (pixels image)
          len          = length message
          header       = encodeLength len [p,q,r]
          (p:q:r:rest) = imgVec
          w            = width image

sprinkle' :: [Pixel] -> ST Sprinkler [Pixel]
sprinkle' [] = return []
sprinkle' (p:ps)
  = do sprinkler <- upd advance
       rest      <- sprinkle' ps
       return     $ f sprinkler p : rest
       where f spr pixel | emptyMsg       = pixel
                         | i `mod` s == 0 = encodePixel pixel char
                         | otherwise      = pixel
                         where emptyMsg   = null (message spr)
                               i          = index spr
                               s          = spacing spr
                               char       = ord $ head $ message spr
```

This isn't the prettiest code, as there is a lot of packing and unpacking that goes on, but the top-level procedure is relatively straightforward.

Given this method for encoding a message in an image, the method for recovering a message from an encoded image is very similar; we can use the same `Sprinkler` type to gather the message. To do so, we define a function, `gather`, that takes an image and returns either a string, or nothing.

This function returns nothing if the image cannot possibly encode a message – this is the case when the length of the message (decoded from the header) exceeds the number of available pixels in the image.

Note that this takes advantage of the fact that the `decodeLength` function described above will return a garbage value even if a length had never been encoded. However, if this garbage value is actually less than the available pixels in the image, then this routine will treat it as a legitimate length and attempt to recover a message, eventually outputting garbage.

The implementation of `gather` is given below. `gather'` is very similar to `sprinkle'`; its implementation can be found in the appendix.

```
gather :: Image -> Maybe String
gather image
  | len > availPix image = Nothing
  | otherwise            = Just $ fst $ run (gather' rest) gatherer
    where imgVec         = concat (pixels image)
          (p:q:r:rest)   = imgVec
          len            = decodeLength [p,q,r]
          gatherer       = (genSprinkler image []){spacing = s, msglen = len}
          s              = availPix image `div` len

gather' :: [Pixel] -> ST Sprinkler String
```

2.3 The PPM Image Format

Now that we have an abstract representation of images (the `Image` type), and a method for encoding and decoding strings in these images (the `sprinkle` and `gather` functions), how can we begin to introduce real image files into the world of this program?

Here we introduce the PPM format [6], which takes the simplistic approach of encoding an image as an ASCII text file. The structure of a "plain" PPM file is given in figure 2.

We can define two functions to convert between PPM files and `Image`s, where PPM files are represented as `String`s.

```
type PPM      = String

imageFromPPM :: PPM -> Image
imageToPPM   :: Image -> PPM
```

The implementation of these functions is straightforward, and is given in the appendix.

Now, we have all of the pieces needed to dive into the world of I/O. But it's rare to see PPM files in the wild, because for instance, a typical 122 kB JPEG files expands into a 5.3 MB PPM! We'd like a way of dealing with other kinds of image files.

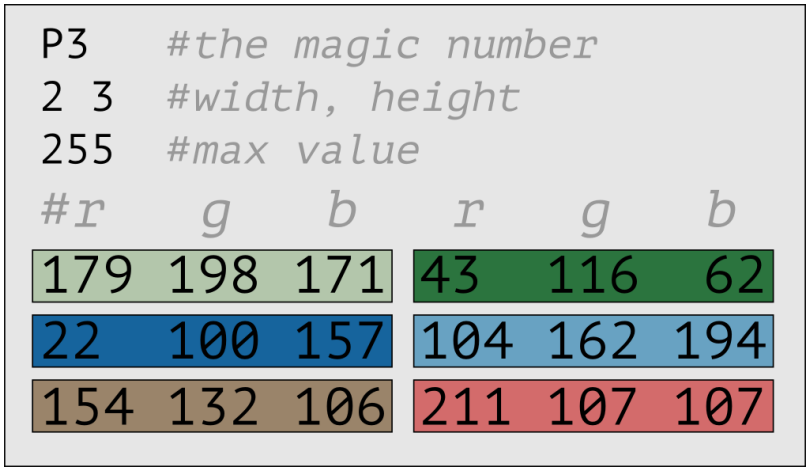

Figure 2: The structure of a PPM file.

2.4 Incorporating ImageMagick

Rather than putting the burden on the user to convert their image files to PPM, we introduce a set of system calls to handle converting between from other image formats to PPM, and from PPM to PNG. These functions are straightforward, but we include them here for readers who may not be familiar with system calls in Haskell.

```
sysConvertToPPM :: FilePath -> IO FilePath
sysConvertToPPM targetFile
 = system ("convert " ++ targetFile ++ " -compress none " ++ ppmFile)
     >>=   handleExitCode
     >>    return ppmFile
     where ppmFile = targetFile ++ ".ppm"

sysConvertToPNG :: FilePath -> IO ()
sysConvertToPNG ppmFile
 = system ("convert " ++ ppmFile ++ " " ++ pngFile)
     >>=   handleExitCode
     where pngFile = takeWhile (/='.') ppmFile ++ ".enc.png"

handleExitCode :: ExitCode -> IO ()
handleExitCode (ExitFailure code)
  | code == notFound = fatalError "You must have ImageMagick installed."
  | otherwise        = fatalError "Image conversion failed."
    where notFound   = 127
handleExitCode ExitSuccess = return ()
```

As a side note, the `-compress none` argument to `convert` is what tells it to generate a "plain" PPM file as opposed to a "raw" PPM, which is not as human-readable.

2.5 The top-level program

Now we are ready to define a top-level `main` function that will take some command-line arguments and do some work:

```
main = getArgs >>= go
  where go                   :: [String] -> IO ()
        go [encoded]          = decodeFile encoded        >> exit
        go [message, target]  = encodeFile message target >> exit
        go _                  = usage                     >> die

encodeFile :: FilePath -> FilePath -> IO ()
decodeFile :: FilePath -> IO ()
usage, exit, die  :: IO ()
```

The definitions of `decodeFile`, `encodeFile`, `exit`, `die`, and `usage` are given in the appendix, along with several other functions for handling errors and managing I/O.

3 Demonstration

A sample execution of the program is shown in figure 3, using Shakespeare's *The Tempest* [5] as the message to be encoded in a sample image.

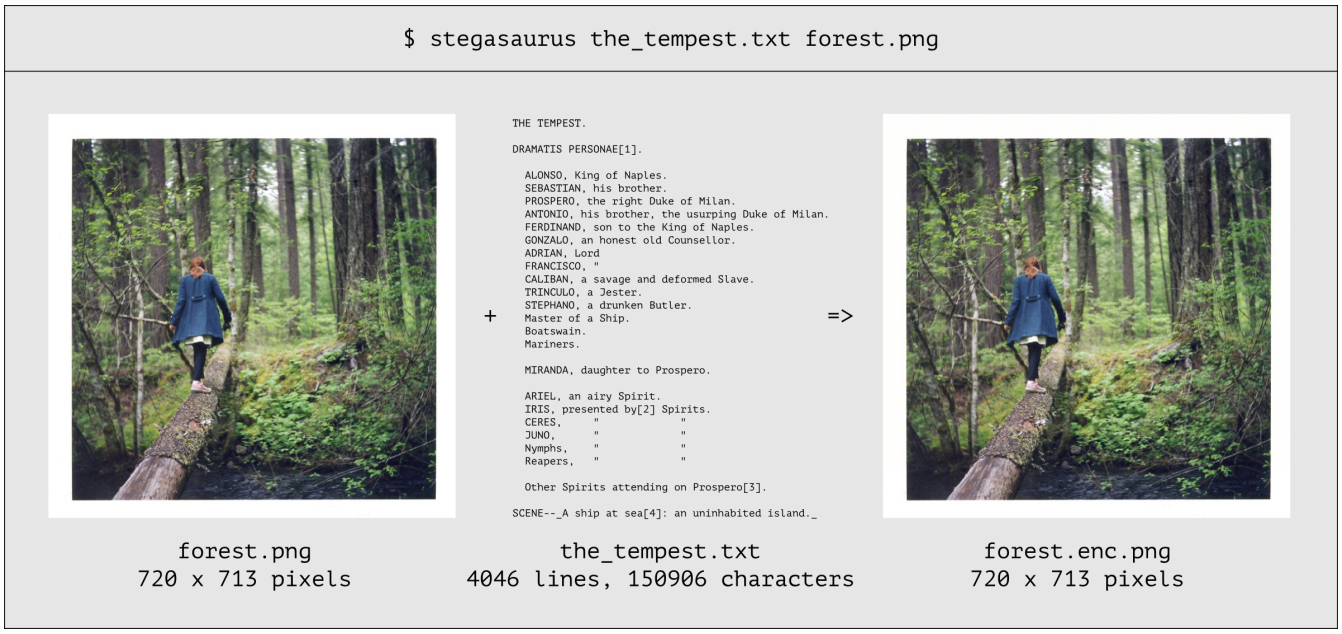

Figure 3: A sample execution of the program being used for encoding.

From the reader's vantage point, these two images should look nearly identical. But a detailed (and contrast-enhanced) view of the upper-right hand corner of the encoded image, given in figure 4, reveals what happens to high-value (for instance, all-white) pixels that are encoded: they pick up a yellowish hue. Why yellow? All-white pixels have all 1s in their components, so any encoding of a value that is not all 1s will necessarily reduce their value. Since three bits from the blue channel are used instead of two (see figure 1), its value is affected more. Blue is the opposite of yellow, so an overall reduction in the blue channel will yield a yellow hue.

The other thing worth noting about this example is that our method of "evenly distributing" pixels yields a striation pattern in the encoded image, which is visible to the naked eye in all-white areas. Knowledge of this pattern, along with the knowledge that the blue channel is affected more than the others, opens

the door for a potential *steganalysis* routine to detect the difference between an encoded and non-encoded image.

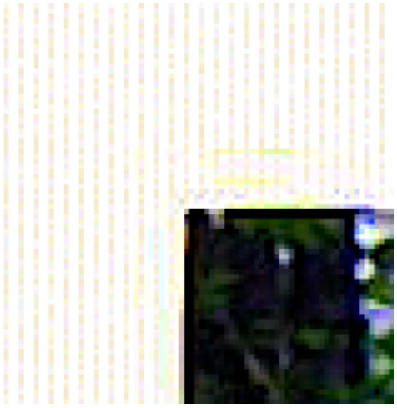

Figure 4: The upper right-hand corner of the encoded image, magnified and contrast-enhanced.

Given an encoded image file, our program can decode it as follows:

```
$ stegasaurus forest.enc.png | head
THE TEMPEST.

DRAMATIS PERSONAE[1].

  ALONSO, King of Naples.
  SEBASTIAN, his brother.
  PROSPERO, the right Duke of Milan.
  ANTONIO, his brother, the usurping Duke of Milan.
  FERDINAND, son to the King of Naples.
  GONZALO, an honest old Counsellor.
```

To show that there is no difference between the original message and the decoded image, we can compare them with the unix `diff` command:

```
$ diff the_tempest.txt <(stegasaurus forest.enc.png)
$ # no output means no difference!
```

But why stop testing there?

4 Testing

This program has a variety of properties we'd like to be sure about. The one tested in the section above ("are the original message and the decoded image the same?") is a good example, but we'd like to be able to test it in more than a handful of sample cases. In this section, we use the QuickCheck library to fuzz-test a variety of predicates we'd like to hold. This is not a substitute for a formal proof of correctness, but it is a step in the right direction.

Note: to avoid a redundant series of lines stating "`+++ OK, passed 100 tests`", we elide the results of each individual test; suffice it to say that all tests in this section pass as expected.

First, we'd like to ensure that if we encode a pixel with an ASCII value n, then decoding it will always

yield the same value *n*:

```
prop_pixelIdentity p
  = forAll asciiVal $ \n ->
    n == decodePixel (encodePixel p n)
    where asciiVal = choose (0, 127)
```

What is an `arbitrary` pixel? It is a pixel for which each component is a random value between 0 and 255:

```
instance Arbitrary Pixel where
  arbitrary
    = do r <- componentVals
         g <- componentVals
         b <- componentVals
         return (Pixel r g b)
         where componentVals = choose (0, 255)
```

We'd also like to ensure that our "header" encoding holds for all valid lengths (between 0 and the maximum value):

```
prop_lengthIdentity
  = forAll triplets $ \t ->
    forAll lengths  $ \l ->
    l == decodeLength (encodeLength l t)
    where triplets = vectorOf 3 arbitrary
          lengths  = choose (0, maxLength)
```

Now we can scale up a bit and begin thinking about whole images. Before we do that, we take a moment to consider what an `arbitrary` image might be. Allowing the height and width to be truly arbitrary integers would yield many gargantuan images, as well as many invalid ones (with negative dimensions). Thus, it makes sense to impose a limit on the maximum size of the images. For ease of testing, we consider this maximum size to be 500×500 pixels.

```
instance Arbitrary Image where
  arbitrary
    = do height <- reasonableSize
         width  <- reasonableSize
         pixels <- vectorOf (height * width) arbitrary
         return $ Image height width (group width pixels)
         where reasonableSize = choose (1, 500)
```

We also define `asciiChar` to generate plain 7-bit ASCII characters (with values between 0 and 127).

```
asciiChar :: Gen Char
asciiChar  = elements (map chr [0..127])
```

Now we are able to test the identity property from the end of section 3 a little more thoroughly:

```
prop_imageIdentity img
  = forAll asciiString  $ \msg    ->
    length msg <= availPix img   ==>
    (sprinkle msg img >>= gather) == Just msg
    where asciiString = listOf1 asciiChar
```

We'd also like to test a couple of negative properties where the length of the message causes the encoding to fail. In the first case, the length of the message exceeds the available pixels in the image. In the second case, the length of the message exceeds the maximum possible length that can be encoded in three pixels

(see section 2).

```
prop_exceedAvailPixFail img
  = forAll longMsg $ \msg ->
    sprinkle msg img == Nothing
    where longMsg = elements [availPix img + 1 .. maxLength]
                      >>= flip vectorOf asciiChar

prop_lengthOverflowFail img =
  forAll overflow  $ \msg ->
  sprinkle msg img == Nothing
  where overflow = vectorOf (maxLength + 1) asciiChar
```

QuickCheck's ability to generate and collect random data can be used for more than just testing. In the next section, we use QuickCheck to help determine how much information can be encoded in an image before it starts to degrade.

5 Analysis

To the naked eye, the sample encoded image generated in section 3 was nearly indistinguishable from its unencoded counterpart. This naturally raises the question – how much information can we hide in an image before it becomes apparent?

5.1 Signals and Noise

In order to answer this question more rigorously, we need a metric for evaluating the difference between an encoded image and its unencoded counterpart. One such metric is the *peak signal-to-noise ratio* (PSNR), which is commonly used for evaluating the performance of image compression algorithms, where a higher PSNR indicates a higher fidelity to the original image. [7].

The PSNR is defined mathematically as follows [8], where P is the unencoded image, Q is the encoded image, P_i is the value of the ith pixel in P, and $\max_i |P_i|$ is the maximum (possible) pixel value in P:

$$\text{PSNR} = 10 \log_{10} \frac{\max_i |P_i|}{\text{MSE}}$$

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (P_i - Q_i)^2$$

This definition applies to grayscale images. For RGB images, the PSNR is the same formula, but the MSE (mean squared error) is the sum of the MSEs for each component, divided by 3.

Given that max_i in our case is 255, these definitions can be expressed in Haskell as follows:

```
componentMSE :: (Pixel -> Int) -> Image -> Image -> Double
componentMSE component p q
  = 1 / fromIntegral n
    * (fromIntegral $ sum $ map (^2) $ zipWith (-) (flatten p) (flatten q))
  where flatten = map component . concat . pixels
        n       = width p * height p

mse     :: Image -> Image -> Double
mse p q  = 1 / 3 * (componentMSE r p q + componentMSE g p q + componentMSE b p q)

psnr    :: Image -> Image -> Double
psnr p q = 10 * logBase 10 (maxI^2 / mse p q)
   where maxI = 255
```

We're interested in examining how the PNSR might change depending on how much of an image is used for hiding a message. We could collect this data manually, but fortunately we can coerce QuickCheck into doing it for us, using its `collect` feature:

```
test_psnr
  = quickCheckWith stdArgs{maxSuccess = 500} $
    forAll arbitrary         $ \img ->
    forAll (usageLevels img) $ \msg ->
    collect (usageRatio msg img) $
    collect (psnr img (fromJust (sprinkle msg img))) $
    True
    where usageRatio msg img
            = ceiling(fromIntegral (length msg) / fromIntegral (availPix img) * 100)
          usageLevels img
            = oneof (map (usageVec img . (/100). fromIntegral) [1..100])
          usageVec img usg
            = vectorOf (floor (fromIntegral(availPix img) * usg)) asciiChar
```

On each test, this code generates an arbitrary image, along with a random string that uses anywhere from 1 to 100% of the pixels image. It collects the usage percentage, along with the PSNR for the image encoded with that string.

Note the use of the trivial property `True`. We're not interested in testing any laws here, just in collecting data. We know the encoding will be successful due to the constraint we put on the message length, but even if that constraint was not there, the collection would abort if the `fromJust` failed, so the use of `True` isn't hiding any potential failures.

Figure 5 shows a graph of data generated by single run of `test_psnr`

The most interesting feature of the graph is the limit around 40 dB – no matter how many characters we push into the image, it never degrades beyond that point. Though it is difficult to translate between PSNR (in dB) and perceived image quality, the typical range for image compression algorithms is 20-40 dB [8]. In general we can conclude that the steganography method we've outlined so far will likely have a minimal perceptual impact, even at 100% usage.

This makes sense, given that we're only using 7 out of 24 bits for our encoding. But we still haven't answered our initial question: how much image information can we lose before it starts to become apparent to the naked eye?

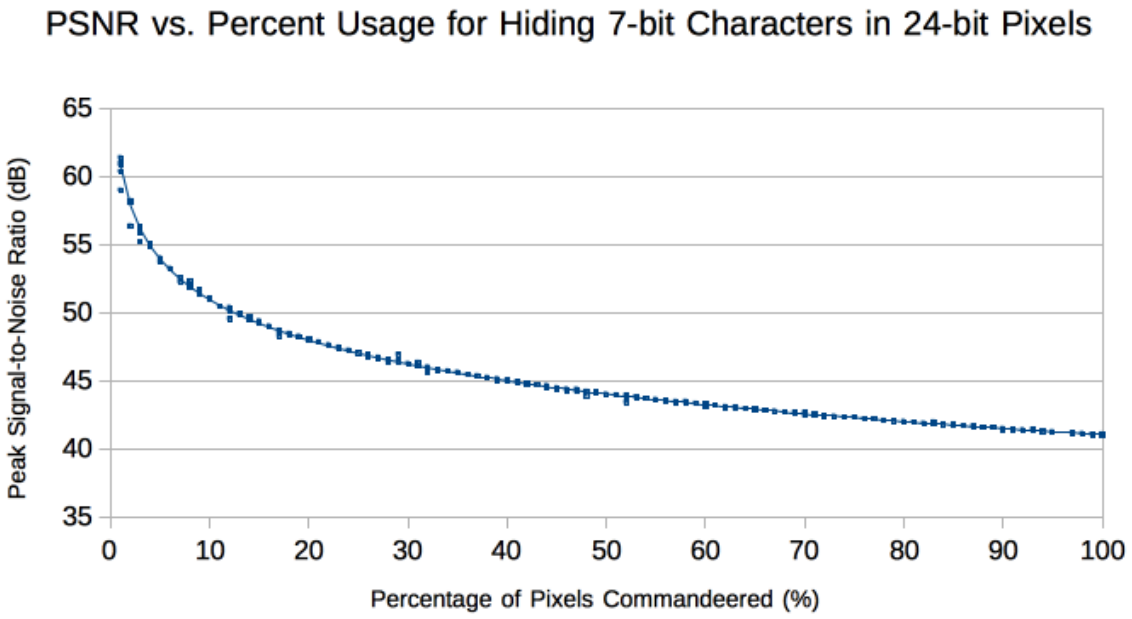

Figure 5: Data collected by the test_psnr function.

5.2 The Limits of Information Hiding

To aid us in exploring this question further, we devise a method for "corrupting" *n* bits of a pixel by overwriting them with noise information:

```
corruptPixel :: Int -> Pixel -> Pixel -> Pixel
corruptPixel nBits (Pixel rNoise gNoise bNoise) (Pixel r g b)
  = Pixel cr cg cb
    where cr     = corrupt r nRed   rNoise
          cg     = corrupt g nGreen gNoise
          cb     = corrupt b nBlue  bNoise
          nRed   = nBits `div` 3
          nBlue  = nRed + if (nBits `mod` 3) > 0 then 1 else 0
          nGreen = nRed + if (nBits `mod` 3) > 1 then 1 else 0
          corrupt component nBits noise
            = (component `shiftR` nBits `shiftL` nBits) .|. (noise .&. mask nBits)
```

This function works from "right" (blue) to "left" (red): if the number of bits to be corrupted is divisible by 3, each component will get its share. If there are extra bits left over, they will go first to blue, then green.

Given a function to corrupt *n* bits of a single pixel, we define a function to corrupt *n* bits of every pixel in the entire image. For simplicity's sake we use black (all zeroes) as the noise value:

```
corruptImage :: Int -> Image -> Image
corruptImage nBits img
  = img {pixels = corruptImage}
    where corruptImage = (map.map) (corruptPixel nBits (Pixel 0 0 0)) (pixels img)
```

With these two definitions, we can abuse QuickCheck in a similar manner as above, and have it collect some data for us:

```
test_corrupt
  = quickCheck          $
    forAll arbitrary    $ \img ->
    forAll corruptLevel $ \cl  ->
    collect cl          $
    collect (psnr img (corruptImage cl img)) $
    True
    where corruptLevel = choose (1,24)
```

This function generates an arbitrary image, and a random number of bits to be corrupted. Note that we leave out 0, because identical images have an undefined PSNR (due to division by zero). Figure 6 shows the graph of data generated by a single run of `test_corrupt`.

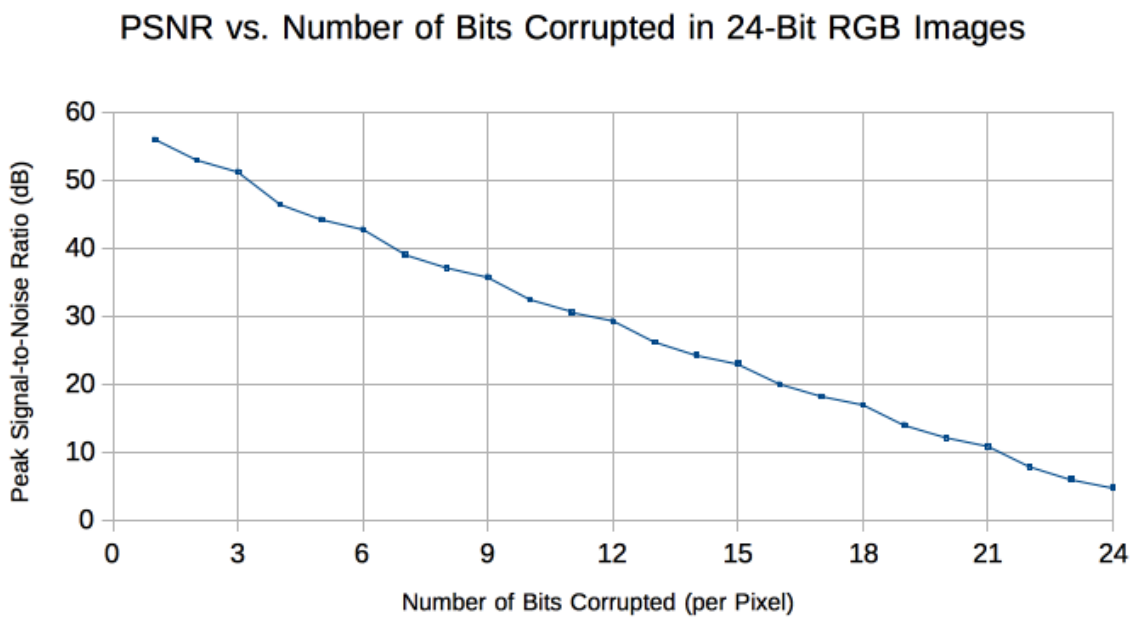

Figure 6: Data generated by the test_corrupt function.

Perhaps the most interesting result of this test is that we could cram twice as much hidden data into the image – up to 14 bits per pixel – and still be within the 20 - 40 dB range that is typical for compression algorithms.

It is also worth nothing that the PSNR for a corruption level of 7 bits is right around 40 dB, which is the limit observed for our steganography method.

Given that decibel data is a little abstract, it is also worth considering visual impact as a function of the number of bits corrupted. This is a little bit more difficult to do with QuickCheck, so we define a set of functions specifically for corrupting an image with *n* bits of random data:

```
randCorruptPixel :: Int -> Pixel -> IO Pixel
randCorruptPixel nBits pixel
  = do rNoise <- posRand
       gNoise <- posRand
       bNoise <- posRand
       return (corruptPixel nBits (Pixel rNoise gNoise bNoise) pixel)
       where posRand = randomIO >>= return.abs
```

```
randCorruptImage :: Image -> Int -> IO (Image)
randCorruptImage img nBits
  = do corruptImageVec  <- mapM (randCorruptPixel nBits) imageVec
       return img{pixels = group (width img) corruptImageVec}
    where imageVec = concat (pixels img)

corruptDemo :: FilePath -> IO ()
corruptDemo ppmFile
  = do img            <- readFile ppmFile >>= return.imageFromPPM
       corruptImages  <- mapM (randCorruptImage img) [1..24]
       let corruptPPMs = zip [1..] (map (imageToPPM) corruptImages)
       sequence_ [ writeFile ("corrupt" ++ show level ++ ".ppm") ppmData
                 | (level, ppmData) <- corruptPPMs]
```

It is worth noting here that `posRand` generates values well outside the normal range (0 to 255) for RGB pixel values, but only the low-order bits of the noise value are used to overwrite the low-order bits of the pixel, so this violation does not matter.

The `corruptDemo` function reads in a PPM file, generates a different version of the image for each corruption level (from 1 to 24), and then writes out each as a new PPM file. A sample of the results generated by `corruptDemo` is given in figure 7.

It worth noting that the image looks unchanged to the naked eye for up to 9 bits of corruption – meaning that 3/8ths of the image data can be used for hiding information, with virtually no perceptible consequences. Only around 12 bits – 1/2 the image data – does the degradation begin to become apparent. Even around 15 bits of corruption, the image looks no worse than one that has been poorly compressed. Beyond that point, the noise begins to overtake the signal and the image becomes lost.

These results have some interesting consequences. For instance, if our steganographic routine used 16 bits per pixel instead of 7, we could encode arbitrary binary files in an image, using two bytes per pixel. Even a diminutive 100×100 pixel color image could hold a binary file up to 20 kB – enough for small executable.

Given that half of the low-order bits can be used for noise with few consequences, it is also conceivable that we could store a second image in those bits, which could be recovered by swapping the high-order 12 bits with the low-order 12 bits. This would have the advantage of being able to recover the hidden message "in-place" without losing the original image – it could be recovered simply by running the decoder a second time.

Of course, the most widespread application of this sort of analysis is not steganography but rather lossy image compression – if we can get away with throwing away half the data without seriously damaging image fidelity, why not do so simply to save space?

In general, we can consider the low-order bits in an image to be a form of noise, for which the high-order bits are the signal; image steganography and lossy compression both take advantage of this by hijacking (or simply removing) the noise.

This yields a key observation, which is that the best media for doing steganography are those with some amount of background noise, in which the information can be hidden without a perceptible impact on the signal. This is why the recent usage of vectors like network traffic have been so effective – there is a large amount of natural noise and error that can be taken advantage of [1]. The next section considers this in

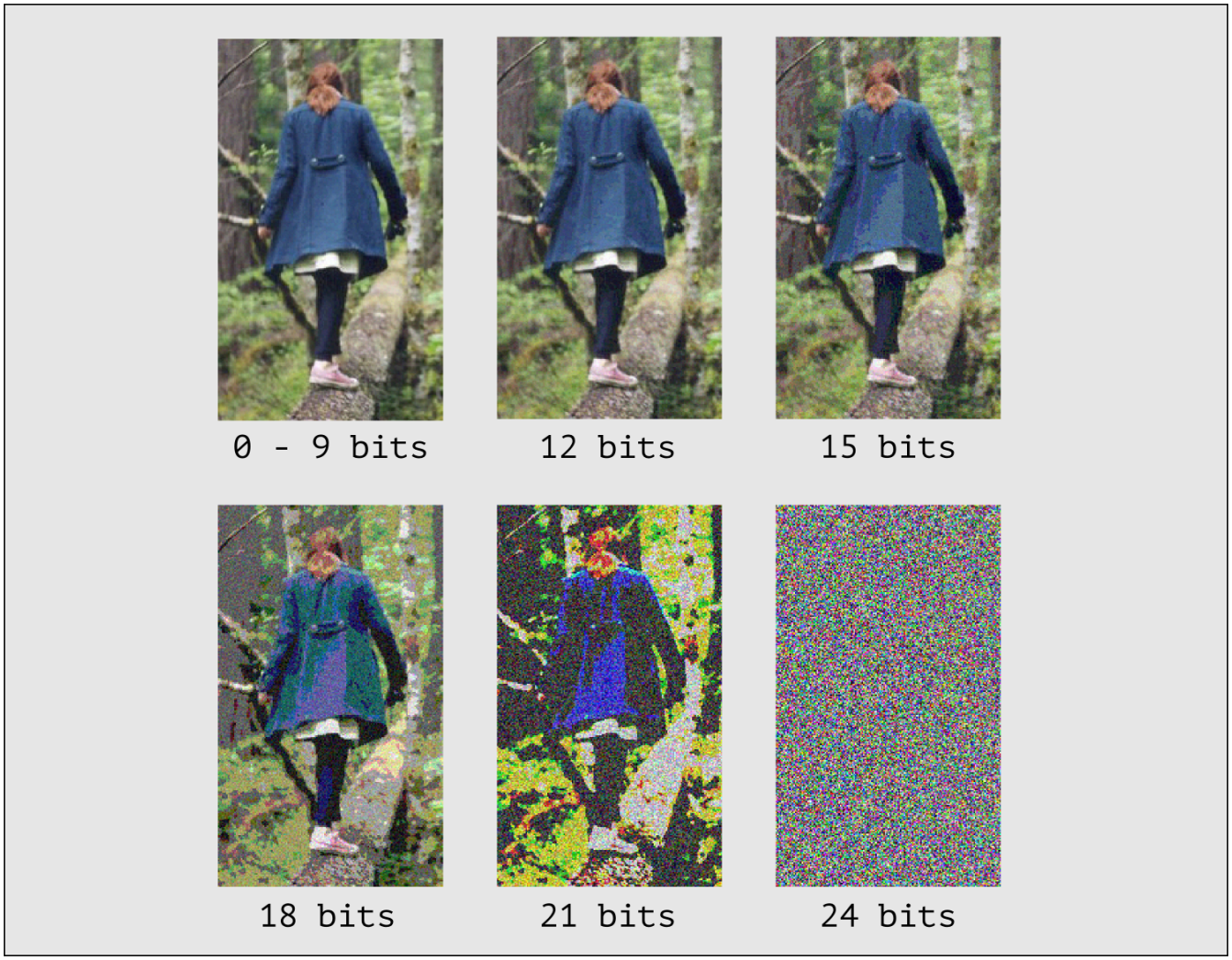

Figure 7: Edited sample of results from "corruptDemo" function.

relation to some generalizations of our steganography program.

6 Further Developments

Consider a hypothetical typeclass, `StegMedium`, denoting the set of types for which a message could be steganographically hidden in a value of that type:

```
type Message = String

class StegMedium a where
  hide   :: Message -> a -> Maybe a
  reveal :: a -> Maybe Message
```

Now, any noisy medium that we can successfully represent in memory could potentially be an instance of `StegMedium`. We can imagine a few such instances:

```
instance StegMedium Image where
```

```
  hide   = sprinkle
  reveal = gather

instance StegMedium Video where
  ...

instance StegMedium FileSystem where
  ...

instance StegMedium tcpPacketStream where
  ...
```

In principle, it would be possible to build up a library of such instances, facilitating the easy development of steganographic tools (provided of course that the libraries existed to define each potential target medium).

This is a fairly high-level goal; in terms of the image steganography routine outlined in this report, there are several low-level improvements that can be made. For instance, the program could be easily extended to work with arbitrary binary files by using one more bit in the encoding. The results demonstrated in section 5 show that this would have a negligible impact on the perception of an encoded image.

Furthermore, the program's reliance on the PPM format and the ImageMagick suite of tools is something of a crutch; a more complex implementation might be able to handle different colorspaces (such as HSL or CMYK) and potentially even lossy formats (such as JPEG).

Regardless, the program is a good first success, and its relatively short and simple implementation may point to one of the reasons why steganography has remained popular – fewer technical skills are required to implement a steganographic protocol than say, a cryptographic protocol, yet steganography may be sufficient for simple covert message passing.

References

[1] Zielińska, Elżbieta, Mazurczyk, Wojciech, and Krzysztof Szczypiorski. 2014. "Trends in Steganography." *Communications of the ACM* 57(3):86-95. doi: 10.1145/2566590.2566610

[2] Rogoway, Mark. 2014. "PSU professor wins surprise, $100,000 grant from Google's Eric Schmidt to help overcome online censorship." *The Oregonian*. March 13. http://www.oregonlive.com/silicon-forest/index.ssf/2014/03/psu_professor_wins_surprise_10.html

[3] Dyer, Kevin P., Coull, Scott E., Ristenpart, Thomas, and Thomas Shrimpton. 2013. "Protocol Misidentification Made Easy with Format-Transforming Encryption." *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* 61-72. doi: 10.1145/2508859.2516657

[4] Jones, Mark P. 2014. Lecture material presented in CS557 at Portland State University, February 27.

[5] Shakespeare, William. *The Tempest.* Project Gutenberg. http://www.gutenberg.org/files/23042/23042.txt

[6] Netpbm. 2013. "PPM Format Specification." Last Modified November 2. http://netpbm.sourceforge.net/doc/ppm.html

[7] Netpbm. 2001. "Pnmpsnr User Manual." Last Modified March 4.

http://netpbm.sourceforge.net/doc/pnmpsnr.html

[8] Salomon, David, Motta, G., and D. Bryant. 2007. *Data Compression: The Complete Reference*, 281-282 London: Springer.

7 Appendix

7.1 Library Imports

```
import Data.Char
import Test.QuickCheck hiding ((.&.))
import Data.Maybe (fromJust)
import Data.Bits
import System.Random
import System.Environment
import System.IO
import System.Cmd
import System.Exit
```

7.2 Pure Helper Functions

```
encodeLength :: Int -> [Pixel] -> [Pixel]
encodeLength int [a,b,c]
  | int > maxLength = error "overflow in length encoding"
  | otherwise       = [encodePixel a one, encodePixel b two, encodePixel c three]
   where one   = (int `shiftR` 14) .&. mask 7
         two   = (int `shiftR` 7)  .&. mask 7
         three =  int              .&. mask 7

decodeLength :: [Pixel] -> Int
decodeLength [a,b,c] = one .|. two .|. three
  where one   = (decodePixel a) `shiftL` 14
        two   = (decodePixel b) `shiftL` 7
        three = (decodePixel c)


genSprinkler :: Image -> String -> Sprinkler
genSprinkler image message
  = Sprinkler{ index   = headerLength,
               spacing = availPix image `div` len,
               message = message,
               currlen = 0,
               msglen  = len }
               where len = length message

advance :: Sprinkler -> Sprinkler
advance spr
  = spr{ index = i + 1,
         message = advance (message spr),
         currlen = incr (currlen spr) }
```

```
        where i = index spr
              s = spacing spr

              advance [] = []
              advance (c:cs) | i `mod` s == 0 = cs
                             | otherwise      = c:cs

              incr           | currlen spr == msglen spr = id
                             | i `mod` s == 0            = (+1)
                             | otherwise                 = id


gather' :: [Pixel] -> ST Sprinkler String
gather' [] = return ""
gather' (p:ps)
  = do sprinkler <- upd advance
       rest      <- gather' ps
       return     $ f sprinkler p ++ rest
       where f spr pixel | msgGathered     = ""
                         | i `mod` s == 0  = [chr (decodePixel p)]
                         | otherwise       = ""
                         where i           = index spr
                               s           = spacing spr
                               msgGathered = (currlen spr
                                              == msglen spr)

group :: Int -> [a] -> [[a]]
group 0 _  = []
group _ [] = []
group n xs = take n xs : group n (drop n xs)

imageFromPPM :: PPM -> Image
imageFromPPM str
  = Image h w $ group w $ pixels $ map read $ drop 4 $ words str
    where w = read (words str !! 1)
          h = read (words str !! 2)
          pixels []         = []
          pixels (x:y:z:xs) = Pixel x y z : pixels xs
          pixels _          = error "Failure loading ppm file"


imageToPPM :: Image -> PPM
imageToPPM img
  =  "P3\n"
  ++ show (width  img) ++ " "
  ++ show (height img) ++ "\n"
  ++ "255\n"
  ++ layout (concat (pixels img))
     where layout :: [Pixel] -> String
           layout = unlines . map unwords . group 18 . concat . map expand
           expand :: Pixel -> [String]
           expand (Pixel r g b) = [show r, show g, show b]
```

7.3 I/O Handling Functions

```
encodeFile :: FilePath -> FilePath -> IO ()
encodeFile messageFile targetFile
 = do message       <- open messageFile
      ppmPath       <- sysConvertToPPM targetFile
      ppmData       <- readFile ppmPath
      sysCleanup       ppmPath
      let maybeEnc   = Just (imageFromPPM ppmData)
                         >>= sprinkle message >>= return . imageToPPM
      case maybeEnc of
        Nothing     -> fatalError "Message is too long for this image."
        Just enc    -> do
          let encPath    = ppmPath ++ ".enc"
          writeFile        encPath enc
          sysConvertToPNG  encPath
          sysCleanup       encPath

      where open "-" = getContents
            open  f  = readFile f

decodeFile :: FilePath -> IO ()
decodeFile encodedFile
  = do ppmPath      <- sysConvertToPPM encodedFile
       ppmData      <- readFile ppmPath
       sysCleanup      ppmPath
       case gather (imageFromPPM ppmData) of
         Nothing    -> fatalError "Not a valid encoded image."
         Just dec   -> putStr dec

exit, die, usage  :: IO ()
exit  = exitWith ExitSuccess
die   = exitWith (ExitFailure 1)
usage = getProgName >>= \p ->
        hPutStrLn stderr
         ("Usage: " ++ p ++ " MESSAGE_FILE TARGET_IMAGE" ++ "\n" ++
          "   or: " ++ p ++ " ENCODED_IMAGE")

fatalError :: String -> IO ()
fatalError e
  = getProgName >>= \p -> hPutStrLn stderr (p ++ ": error: " ++ e) >> die

sysCleanup     :: FilePath -> IO ()
sysCleanup file = system ("rm " ++ file) >> return ()
```